統合並列処理向けの多項式計算ソフトウェアの試作 (数式処理における理論と応用の研究)

AUTHOR(S):

村尾, 裕一; 藤瀬, 哲朗

CITATION:

村尾, 裕一 ...[et al]. 統合並列処理向けの多項式計算ソフトウェアの試作 (数式処理における理論と応用の研究). 数理解析研究所講究録 1999, 1085: 185-197

ISSUE DATE:

1999-03

URL:

http://hdl.handle.net/2433/62793

数理解析研究所講究録
1085 巻 1999 年 185-197

統合並列処理向けの多項式計算ソフトウェアの試作

東京大学大型計算機センター 村尾裕一 (Hirokazu MURAO)
三菱総合研究所 藤瀬 哲朗 (Tetsuro FUJISE)

1 はじめに

筆者らは、大規模な並列計算機を活用するための技術開発と記号処理を中心とした高性能計算機のより広い応用分野の開拓とを目指し、研究開発プロジェクト「統合並列処理技術の開発」(代表者:近山 隆 (東京大学工学系研究科) を 2 年程前に開始し、当初計画したソフトウェアの開発と共に、1998 年の前半に一応の終了へとこぎつけることができた。並列処理に関するプロジェクト全体の計画概要は、成果報告書 ([4] に置かれている) で述べられており、また、その一端であり本稿の主題であるソフトウェアに関しての当初の計画内容は [7] で説明したとおりである。本稿では、プロジェクトの一環として開発を行ったソフトウェアの内の数式処理機能に関して、ソフトウェアの構成や開発時に用いた技法等を簡単に説明する。

1.1 データ並列とタスク並列

並列処理には、大きく分けて、データ並列処理とタスク並列処理の 2 つの形態がある。それぞれの特徴を表 2 にまとめた。データ並列処理は、ベクタ計算機上での処理に代表されるように、数値計算の分野で研究と開発が進められ実用上不可欠となっている一方、タスク並列処理に関しては、記号処理を中心に、未だ実験的な段階にある。統合並列処理とは、その両方を統合し、ベクタ並列計算機や大規模な並列計算機のように、両形態を融合して用いるべきハードウェアを活用するための計算環境を構築するものである。

1.2 数式の計算における並列処理

数式処理においても、様々な計算で並列処理が可能である [6]。

データ並列処理向きの数式計算としては、密な多項式の算術演算 (但し、係数は、一語長に収まる程度の大きさの剰余数の場合で、一般的な (多倍長のように構造を持った) 係数の場合はダメ)、フーリエ変換を用いた多項式の乗除算 (=数値評価と補間による決定に他

表 2: データ並列処理とタスク並列処理の特徴

	データ並列処理	タスク並列処理
計算の対象と特徴	規則的なデータ列 (密行列等) 定型的処理	不規則なデータ列や 非定型的処理
計算量の予測 (粒度) 負荷の分散	実行前に確定 小さく効率的に 実行前に決定	実行時まで不確定 分散し易い程度に大きく 実行中に動的に
通信パターン 必要な通信	実行前に確定 同期通信	実行時まで不確定 非同期通信
自動並列化	実用段階 (並列化コンパイラなど) 自動ベクトル化	研究途上 (一部を除き非効率)

ならない)、modular composition: $f(g(z)) \bmod h(z)$ (行列乗算を用いた漸近的高速アルゴリズム (Brent と Kung))、多項式を決定 (補間) するための数値データの生成 (中国剰余定理:剰余 → 任意精度の整数値 + 補間 : 整数値 → 多項式) などが好例である。いずれの場合も、途中の計算は数値処理とみなすことができる。

一方、タスク並列処理向きの数式計算は?といえば、数式処理に現れるような複雑なアルゴリズムでは、複数の独立な部分的な処理群の連なりとなっているのが当たり前であるから、タスク並列処理の対象はいくらでもあるということはできる。しかしながら、... 並列処理をすれば必ず速くなるというものではない! 単純なタスク並列化を行うと、一部の処理が全体の計算時間の大部分を占めてしまうということが起きるためである。その典型的な例が Buchberger 算法の直接的な並列化で、算法自体には高い並列度が認められるものの、その部分について並列処理を施したとしても、計算時間の多くは多倍長整数の計算にばかり費やされるために決して速くはならないというのは良く知られた事実である。つまり、効率の良いタスク並列処理を行うには、等価な部分問題へと分解/分割し、実行時の動的負荷分散を旨く計画する必要がある。特に、計算負荷の分散 (多くのプロセッサを) と通信コストの低減 (少ないプロセッサで) とでは、相反する要素があり、トレード・オフとなる。さらに、利用可能なプロセッサは有限台数であることや通信コストが無視できないことなどが要因となり、現実の計算機システム上で効率の良い並列処理を行うことは容易ではない。

従来から指摘してきたように [5]、多項式因数分解は、データ並列処理が可能な様々な多項式演算を含むと同時に、計算処理量の予測がある程度可能で並列処理可能な複数のタスクへと分割することができ、我々の統合並列処理の研究に適した実験題材である。以降で

は、そのインプリメントにあたり、どのような並列性に着目し、どのようにして実現したかについて説明する。

2 実現法について

2.1 ターゲットとする計算機環境と実現方式

ソフトウェアに関しては、できるだけ既存のものを使い、効率上の問題が生じないように旨く組み合わせて用いることとした。具体的には、次のとおり。

- タスク並列処理のベースとして並列論理型言語処理系 **KLIC** を利用
- 対象システムの特徴に応じたデータ並列機構をプラグイン、統合環境を形成
  - ハードウェアの差異の吸収 (共有メモリ, 分散メモリ, ベクタ型 など)
  - システムソフトウェアの差異の吸収 (並列動作プロセサ群の見え方 など)

  → システムごとに若干異なる実装技術を開発
- プログラミング言語として見ると:
  - 非決定的計算, 非同期通信等のタスク並列制御部は KL1 言語
  - その他は、 C 言語とデータ並列計算のためのパッケージなど

ソフトウェアに関して、上のような構成法をとり、また、同一プロジェクト内でのKLICの新たなハードウェアへの移植が行なわれた結果、並列型ベクタプロセッサまで含めた、あらゆる種類の並列計算機システム上で動作可能なソフトウェアが簡単に開発することが可能となった。実際、下に示した並列計算機種を用いて開発・テストを行った。

並列型ベクタプロセッサ: FACOM VPP
分散メモリ並列機: Sun Ultra Enterprize, NEC Cenju3
共有メモリ並列機: COMPAQ PC サーバ/Linux
(+ 仮想共有メモリ並列機)

2.2 *parallel polynomial factorizer* の構成

図 1 に、我々が開発したソフトウェアの構成を示す。図中の v や p のマークは、各プログラム部位におけるベクタ処理と並列処理の可能性を示す。以下に、ソフトウェアの特徴をまとめる。

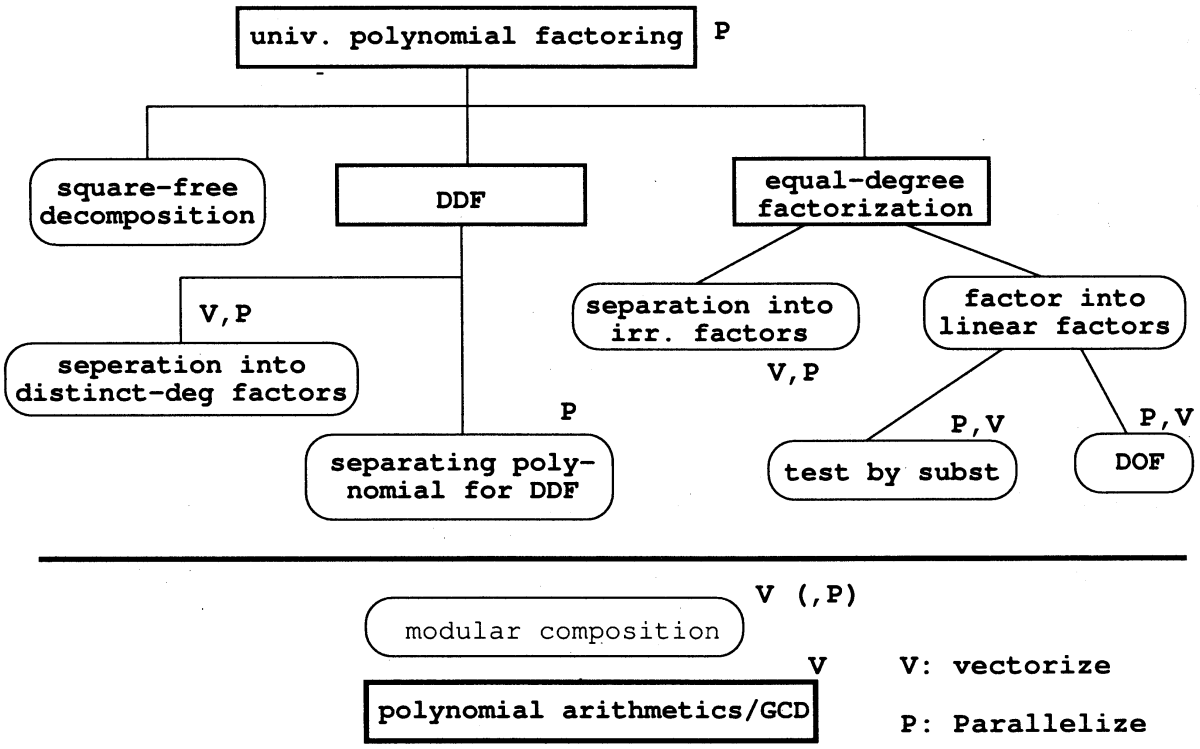


図 1: 並列一変数多項式因数分解計算系

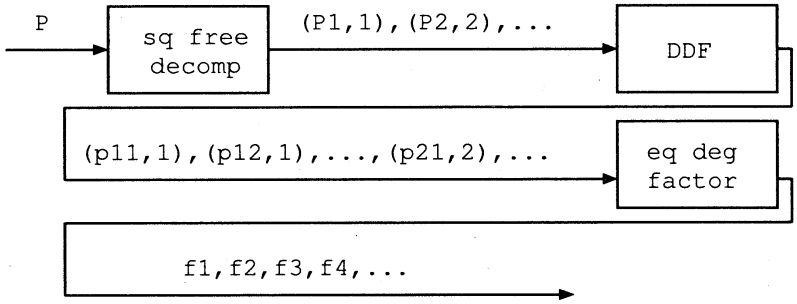
- 主な機能は、$\mathbf{Z}_p$ 上の一変数多項式の因数分解
- 最新のアルゴリズムを利用: von zur Gathen と Shoup のアルゴリズム [3] や Kaltofen と Shoup のアルゴリズム [2] を、漸近的高速アルゴリズムの利用可能性を検討した上で、適宜組み合わせて利用。
  - 無平方分解：通常の逐次アルゴリズム (注:係数は有限体であるので、そのための対処が必要)
  - 次数別因数分解 (DDF): coarse DDF + fine DDF (3.2 節で概説)
  - 同次数因数分解: 数種類の分離多項式の形

  いずれにおいても、主な仕事は、**GCD 計算の繰り返しによる因子の分離 = 代数的探索の処理である。**
- KLIC + C/ FORTRAN(77) で記述: FORTRAN はベクタ (データ並列) 処理用 (多項式の算術演算で多用。DFT 利用の場合も)

3 多項式因数分解における並列性

3.1 パイプライン並列

前節で述べたとおり、多項式因数分解のアルゴリズムは、主な3つの分解の段階から成り、多項式は少しずつ分解されて行って最終的には既約因子にまで分解される。アルゴリズムを記述する場合には、これらの3つの段階を明確に分けて書くのが普通だが、言うまでもなく、各段階で分離された因子多項式は、同じ段階で得られる因子の全てが揃うのを待つまでもなく、次の分解の段階へ進むことができる。つまり、この3つの段階から成るアルゴリズム全体の流れはパイプライン的であり、分離された因子が得られ次第次のステップへと行けるような経路だけを用意してつないでおけば、各段階はパイプライン並列的に処理を進めることができる。但し、実際にそのような並列実行を行うかは、スケジューリングの問題によって決まる。



3.2 Coarse and Fine DDF

無平方な多項式を F とし、F の既約因子で次数が i に等しいものの積を f_i で表せば、次数別因数分解 (DDF) とは F を $\{f_i\}_i$ へと分解することである。最近の DDF アルゴリズムは、大雑把に分解する coarse DDF とそれ結果得られた因子 (partial factorization) を次数別の因子にまで細かく分解する find DDF の 2 つの段階からなる。即ち、整数 l を $\sqrt{\deg(F)/2}$ 程度の値とし、$F_{k+1} = \prod_{i=1}^{l} f_{kl+i}$ と定義すれば、coarse DDF では F を $\{F_k\}_k$ へと分解し、fine DDF では F_k を $\{f_{kl+l-i}, 0 \leq i < l\}$ へと分解する。それぞれの分解 (因子の分離) は、適当な補助多項式との GCD 計算の繰り返しにより行う。

- coarse DDF: $F_k \leftarrow \gcd(F, \Lambda_k)$, $k = 1, 2, ...$
  繰り返し時には、分離された因子は除去 ($F \leftarrow F/F_k$) していく必要があり、よって逐次性が必要である。

- find DDF: $f_{kl-i} \leftarrow \gcd(F_k, \lambda_{kl} - \lambda_i)$, $0 \leq i \leq l-1$.
  $k = 1$ の場合のみ、coarse DDF と同様に逐次性が要求される。

これらの処理の中には、次の 3 種類の代数的独立性が存在し、並列処理が可能である [1]。

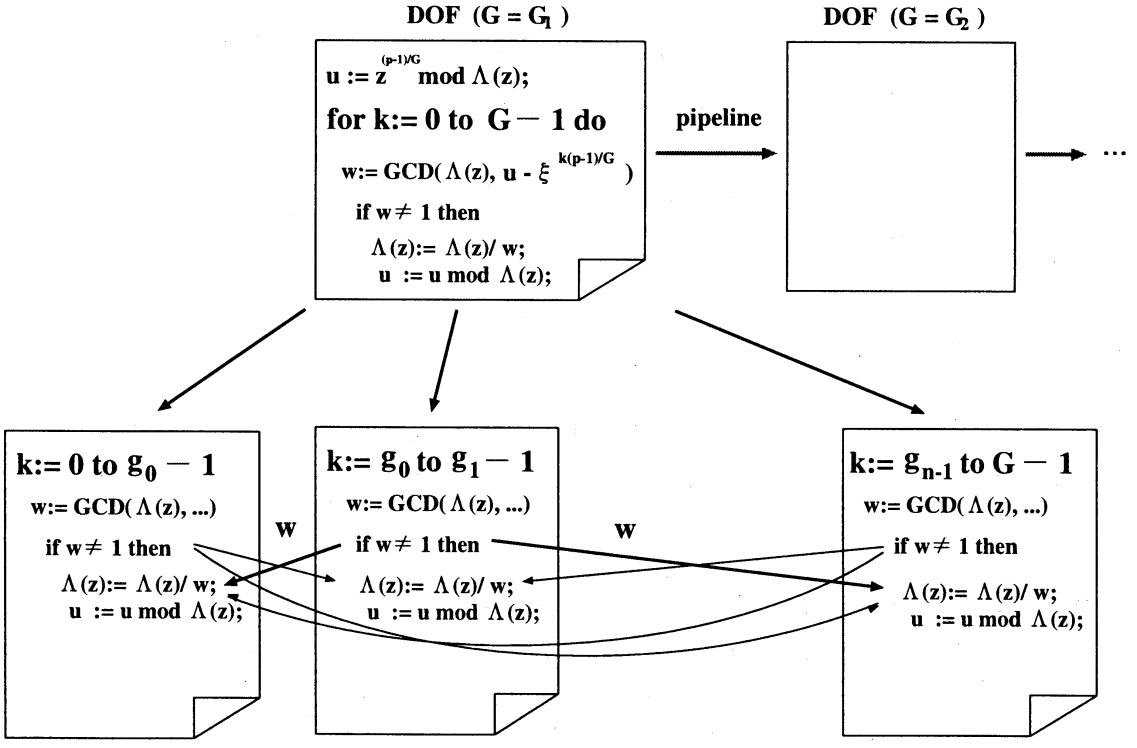
- coarse DDF(の GCD 計算の繰り返し) と fine DDF とは独立。
- 複数の fine DDF は互いに独立。
- $k > 1$ の場合の fine DDF では、異なる i の値に対する GCD 計算は代数的に独立。

3.3 並列探索

線形多項式の積 f_1 の既約因子への分解は、$f_1 = 0$ の根 $\in \mathbf{Z}_p$ の探索に他ならない。この因数分解の方法とその並列処理については、MF@JSC96 で述べたとおりである。同様の方法は、一般的な同次数因数分解 (f_i の既約因子への分解) にも適用可能である。即ち、次数の等しい既約多項式の積 f を分解するには、適当な多項式 B を用いて、次式に基づく GCD 計算の繰り返しにより因子を分離する。

$$f \to \prod_{k=0}^{G-1} \gcd(f,\ B^{k(p-1)/G} - \xi^{k(p-1)/G})$$

ここで、整数 G は $p-1$ の適当な因子とする。異なる k の値に対する GCD 計算は代数的には独立であり、k の値の範囲を分割して並列処理をすることが可能である。その場合、繰り返し計算の途中 (の k のある値) で見つかった因子は、他の k の値で見つかることはあり得ないのでその分を除外する (f を除する) ことができる。これは、探索の対象を減らすという意味で代数的な「枝苅り」とみなすことができ、必ずしも必要な処理ではない。よって、実際には、通信コストとの兼ね合いで行うか否かを決めることになる。



なお、d 次の既約多項式の積である f_d の分解に用いる分離多項式としては、条件 $B^{p^d} \equiv B \bmod f_d$ を満たす多項式 B が利用可能であり、我々は、インプリメントの容易さと経験から Ben-Or separator を用いることとした。即ち、乱数的に生成した多項式 A と整数値 $\alpha \in \mathbf{Z}_p$ に対し、$B = T_d(A) + \alpha$ とする (但し、T_d は McEliece: $T_d(z) = z^{p^0} + z^{p^1} + \cdots + z^{p^{d-1}}$)。

4 KLIC プログラミングと技法

4.1 基本的な実行機構

KLIC のプログラムは、ゴールの述語の true への書き換えと変数の unification の繰り返しとして実行が行われる。下図の例は、我々が今回試作した factorizer のトップレベルの述語 factor を定義する節である。述語 factor は、多項式 GPpol と素数 Prime を引数として受け取り、因数のリストを Facs にユニフィケーションし、結果の値として返す。

```
factor( GPol, Prime, Facs ) :-
    factor_init( GPol, Prime, Pol, PInfo ),
    factor_square_free( Pol, PInfo, Lsqfr ),
  % Lsqfr をストリーム (パイプライン) として利用: 要素としてデータを流す
    factor_Lsqfr( Lsqfr, PInfo, Lans ),
    factor_Lpol_to_gpol( Lans, Facs ).
```

因数分解そのものは、右辺の 4 つの述語 factor_init (初期化), factor_square_free (無平方分解), factor_Lsqfr (リストの要素として与えられる無平方因子の因数分解) 及び factor_Lpol_to_gpol (入力と同じデータ構造への変換) を新たなゴールとすることで処理が進められる。ここで、処理の方法に関し、次の 2 点に注意。

- ゴールの実行の順序は規定されておらず、複数のゴールを同時 (並列) に実行してもよい。
  言語処理系で決められた処理の可能性の結果として、括弧内に記した処理内容の逐次性は保たれる。

- リスト構造を、ゴールを実行する 2 つのプロセスを繋ぐ通信路のストリームとし、一方から順に出力されるデータの列を他方への入力とすることが可能。

これらのお蔭で KLIC では、3.1 節で述べたパイプライン的な並列処理が自動的に実現される。

ストリームに対し、入力のデータ列を加工しては出力するフィルタとして働くプロセスは、次のように定義する。ここで、入力が空か (尽きたか) 否かの判断は定義の右辺にガー

ドを置くことにより行い、一つの述語内での条件分岐が実現される。factor の右辺として実行すべきゴールに加えられた factor_Lsqfr は、その第一引数が空 ([]) であるか、少なくとも先頭の一つの要素 [M,Pm] が確定したリストとなった時点で (リストの全要素が確定している必要はない)、そのボディのゴールの実行に移る (ゴールを実行すべきゴールの集合に加える)。

```
factor_Lsqfr( Lsqfr, PInfo, Lans ) :- Lsqfr = [] | Lans = [].
factor_Lsqfr( Lsqfr, PInfo, Lans ) :-
    Lsqfr = [ [M,Pm] | LsqfrR ] |              ← データの取り出し
        factor_ddf_control( Pm, PInfo, Lirr ),
        factor_merge_MthLirr( M, Lirr, Lans, LansR ),
        factor_Lsqfr( LsqfrR, PInfo, LansR ). ← 他のデータの処
理へと再帰
```

4.2 C(や FORTRAN) の利用

KLIC では、処理系内部に関する知識をある程度有するプログラマを対象として、

- C を用いて任意のデータ構造を追加するための generic オブジェクトの機構と
- KLIC プログラム中に、C 言語によるプログラムを節のガード部に埋め込むinline の機構

が用意されている。これらの機構を駆使すれば、KLIC での記号処理指向の処理に適さないような処理を、C 言語 (や、それを通しての Fortran 等の他言語) によってプログラムすることが可能である。我々は、多項式の算術演算などの単純な数値処理にこの機構を利用し、(KLIC によらない) 外部のプログラムにおいてデータ並列処理を実現している。

```
factor_univpol_plus( [D1,CL1], [D2,CL2], [Prime|_], C, Ans ):-
    inline:"{
      unsigned int d1 = intval(%0), d2 = intval(%2),
                   p = intval(%4), d;
      ModNum *c1 = ihv2i(%1), *c2 = ihv2i(%3), *c = ihv2i(%6);
            % ↑ 多項式の係数ベクタ
        d = MUP_plus( d1, c1, d2, c2, p, c );
        % C/FORTRAN で書かれた外部サブルーチンの呼び出し
        %5 = makeint(d);
    }":[ D1+int, CL1+object, D2+int, CL2+object, Prime+int,
          Deg-int, C+object ]
            % ↑ inline 部とのデータの交換:授受 (+, -) と型
        | pol_or_int( Deg, C, Ans ).
```

この例は、2 つの多項式の和を (ガード部で) 計算する節で、inline 部とのデータの授受は、`inline:"...":`の後に書かれた引数の列「[引数 ± 型, ...]」(+ 或は − は、inline 部がデータを受け取るのか値を返すのかの指定) と inline の C プログラム中の「%n」の指定 (n は、対応する引数の位置) によって実現されている。即ち、多項式の次数と係数の配列を +型で宣言された引数として inline 部に渡し、計算結果の多項式の次数と係数の数値列を各々 -int で宣言された変数 Deg と C+object で渡した領域で受け取っている。

4.3 並列処理の制御 − 優先度の指定

本節冒頭の 4.1 節でも触れたとおり、並列実行可能な環境における KLIC プログラムの実行は、複数ゴールの並列実行という形で自動的に並列処理が行われる。これは、待ち行列にたまった実行可能なゴールの処理を、空き状態となったプロセッサが順次行っていき、その結果、複数個のゴールの実行が同時に行われるということに過ぎない。これには、負荷の分散が自動で動的に行われるという利点もあるが、その処理の並列性は、3 節で述べたようなアルゴリズムに内在する並行性とは無関係である。また、アルゴリズムには、幅優先や深さ優先といった、処理全体の内のどの部分を先行させるかという概念が、並列性や並行性とは独立に存在する。KLIC では、優先度を、プラグマを用いてプログラム中で宣言しておくことにより、こうした実行形態の制御を行うことが可能である。

- 再帰的なプログラムにおける、深さ優先の実行

```
factor_Lsqfr( Lsqfr, PInfo, Lans, NPE ) :-
    Lsqfr = [ [M,Pm] | LsqfrR ] |
        factor_ddf_control( Pm, PInfo, Lirr, NPE ),
        factor_merge_MthLirr( M, Lirr, Lans, LansR ),
        factor_Lsqfr( LsqfrR, PInfo, LansR, NPE )
                        @lower_priority.
```

この factor_Lsqfr という述語は、リストの要素として与えられた無平方な多項式の (既約因子への) 因数分解を行うためのプログラムで、リスト中の各多項式に対する因数分解処理は独立なので並列に実行することが可能だが、上の例では、優先度を指定して、一つの多項式の分解が完了してから次の多項式へと移るように (つまり、深さ優先となるように) 実行を制御している。これは、一つの因数分解処理の中だけでも様々な並列処理が可能なので、処理が複雑化し過ぎないようにするためである。

- 多数のプロセッサ上へと広がり過ぎないように

  次の例は、3.3 節で述べた並列処理のパイプラインの部分のプログラムである。即ち、factor_eqdegfac_random_G からの出力 (既約ではない因子多項式) を、ストリームのリスト Lfacs を通して factor_eqdegfac_Lrandom_G へと流している。ここでも上の場合と同様、複数の多項式の処理を同時に進めることよりも、一つの多項式を既約因子にまで分解することを優先させている。

```
factor_eqdegfac( F, DD, PInfo, VL1, VLL, Lirrs, Com ) :-
    ...
    factor_eqdegfac_random_G( F, DD, PInfo, G, N, Alpha,
            VL1, L, VLL, Lirrs1, [], Lfacs, [], Com1 ),
    factor_eqdegfac_Lrandom_G( Lfacs, DD, PInfo, G, N,
            L, VLL, Lirrs, Lirrs1, _, [], Com2 )
                        @lower_priority.
```

4.4 並列処理の制御 – 実行する PE の指定

KLIC には、並列計算機環境で実行する場合に、個々のゴールに対し実行すべきプロセッサ・ノードを指定し、明示的に負荷分散を行う方法が用意されている。3.3 節で述べた方法において k のとりうる値の範囲を分割して単純な並列処理を行うという場合のように、

負荷がほぼ均等となるように処理を分割した上でノードへの分散を行えば、主要なデータの移動 (プロセッサ間の通信) を減らすことができ、実行の効率を向上させられる。次の例は、同次数因数分解における並列探索において、 k の値の範囲をStep 毎に分割し負荷分散を行っているプログラムの一部である。

```
feqdf_G_fork([PE|PERest], F, DD, PInfo, ZtoN, K, Step, G, N, Alpha,
              Com, Stop):-    ...
    wait(PE), K+Step < G, K1:= K - 1, K2 := K+Step |
        Com = {Com1,Com2},          ← 通信用のストリーム。どこかでマージされる
        ...
    % 次式 (for k = K to K2−1 に対応) の実行を node 番号 PE のプロセッサ上で
        factor_eqdegfac_G_exec( F, DD, PInfo, ZtoN, K, Ak_1, K2, N,
                        Alpha, PE, Com1,Stop ) @node(PE),
        feqdf_G_fork(PERest, F, DD, PInfo, ZtoN, K2, Step, G, N, Alpha,
                        Com2, Stop).
```

4.5 PE マネージャー

一般的な (記号) 計算処理においては、3.3 節の場合のように同等の計算量の処理に分割できることは稀であるし、また、実際に実行した場合の処理量が均等になることも必ずしも期待はできない。実際、前節に示したプログラム例でも、分離される因子が特定のノードに偏在して見つかれば、ノードによって処理対象の多項式の次数、ひいては処理に要する計算量が異なることとなる。そのような場合、前節に示したような明示的な方法だけで負荷分散を行っていると、多くのプロセッサがアイドル状態にあるという状況が生まれかねない。我々は、このような並列処理において最も嫌われる状況を避けるために、PE(ノード) の管理をするプロセス (マネージャー) を用意し、ゴールの実行を暇なノードに動的に割り付けるという方法を実現している。処理 (負荷) の分散を司る親のプロセスは、このマネージャーに対し必要な台数のノードを要求し、子のプロセスの各々には、マネージャーから返されたノード番号または後で unification されるノード番号へのエントリのいずれかを付加して新たなゴールを生成する。子のプロセスは、@node() を指定して、与えられた番号のノードで実行を行い、実行終了時にそのノードをマネージャに返却する。ここで、各ノードで実行する子のプロセスは、ある程度の大きさの粒度を持たせるようにするのは言うまでもない。

- PEの割り当て要求と解放通知を送るための専用のメッセージ・ストリーム: pe_manager が管理

```
pe_manager( [ Mesg | StrmR ], ... ) :-
    % PE の割り当てと解放を行う
    pe_manager( StrmR, ... ).
```

- pe_manager が受け取るメッセージの種類

  get_all_pes(*PE_list*, *Nalloc*) : 実行中のプログラムで利用可能な全 PE の割当て。

  get_some_pes(*PE_list*, *n*, *Nalloc*) : *n* 台の PE の割当て。

  PE_list: 割り当てられた PE のノード番号のリスト、
  Nalloc: 割り当てられた PE の台数。

  (注意) *PE_list* には、ユニフィケーションされていない要素も含まれる。そのような要素 (仮想 PE) は、queue(マネージャー中で差分リストとして実現) で管理され、後で解放された PE の番号が割り当てられる。

  put_pe(*n*) : ノード番号 *n* の PE を解放し pe_manager に返す。

5 まとめ

- 統合並列処理技術を利用可能な記号処理環境が新たにできた。
  (ベクトル処理をひとつのタスクとして、タスク並列処理を行うようなことも可)
- 並列処理を行う多項式因数分解の機能が準備できた → 様々な並列処理の形態の実験が可能。
- 算法の設計やチューニング (動的な負荷分散など) の実験材料に (特に、粒度が中くらいの場合の)。

謝　辞

本研究の一部は、情報処理振興事業協会 (IPA) が実施する創造的ソフトウェア育成事業の「統合並列処理技術の開発」(代表者: 近山 隆 (東京大学工学系研究科)) として行なわれました。

参 考 文 献

[1] FUJISE, T., AND MURAO, H. Parallel distinct degree factorization algorithm. In *Proc. ISSAC '96* (1996), Y. N. Lakshman, Ed., pp. 18–25.

[2] KALTOFEN, E., AND SHOUP, V. Subquadratic-time factoring of polynomials over finite fields. In *Proc. 27th Annual ACM Symposium on the Theory of Computing* (Las Vegas, Nevada, May 29–June 1 1995), pp. 398–406.

[3] VON ZUR GATHEN, J., AND SHOUP, V. Computing Frobenius maps and factoring polynomials. *Computational Complexity 2* (1992), 187–224.

[4] 情報処理振興事業協会. 創造的ソフトウェア育成事業. `http://www.ipa.or.jp/`.

[5] 村尾. $\mathbf{Z}_p$ 上の多項式の因数分解 — 高速化技法・ベクトル処理・並列処理 —. 講究録 *920*「数式処理における理論とその応用の研究」. 京都大学数理解析研究所, 8 月 1995.

[6] 村尾, 藤瀬. 数式処理と並列処理. 情報処理 *39*, 2 (1998), 116–121. 特集:数式処理の最近の研究動向.

[7] 藤瀬, 村尾. 一変数多項式因数分解のための並列計算系について. 講究録 *1038*「数式処理における理論とその応用の研究」. 京都大学数理解析研究所, 11 月 1998, pp. 1–6.